Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии» (ИУ7)

Отчет по лабораторной работе №1 по курсу «Анализ алгоритмов»

«Расстояние Левенштейна и Дамерау — Левенштейна»

Группа: ИУ7-53Б

Студент: _____ Дьяченко А. А.
(Подпись, дата) (Фамилия И. О.)

Преподаватель: _____ Строганов Д. В.
(Подпись, дата) (Фамилия И. О.)

Москва, 2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной лабораторной работы является изучение расстояния Левенштейна и Дамерау — Левенштейна.

Расстояние Левенштейна (редакционное расстояние) и его модификация - расстояние Дамерау — Левенштейна, представляют собой метрики, используемые для измерения различий между двумя последовательностями символов. Они определяют минимальное количество односимвольных операций (вставка, удаление, замена и транспозиция), необходимых для преобразования одной последовательности символов в другую. Эти метрики были разработаны советским математиком Владимиром Левенштейном в 1965 году и модифицированы впоследствии с учетом операции транспозиции символов, получив название расстояния Дамерау–Левенштейна.

Расстояние Левенштейна и его модификация имеют широкий спектр применений в различных областях, включая компьютерную лингвистику (автозамена, исправление ошибок), биоинформатику (анализ генома, белковых последовательностей).

Задачи лабораторной работы:

- реализация алгоритмов с использованием динамического программирования;
- сравнение требуемого времени выполнения в тактах процессора и занимаемой памяти;
- подготовка отчёта по лабораторной работе.

1 Аналитическая часть

Расстояние Дамерау — Левенштейна представляет собой метрику, измеряющую разницу между двумя строками, путем подсчета минимального числа операций, необходимых для преобразования одной строки в другую. Эти операции включают в себя вставку, удаление, замену символов и дополнительную относительно расстояния Левенштейна операцию, которая называется транспозицией, которая представляет собой перестановку двух соседних символов в строке. [1]

Обозначения редакторских операций:

а) I - вставка символа (insert);

б) D - удаление (delete);

в) R - замена (replace);

г) M - совпадение двух символов (match);

д) T - транспозиция двух соседних символов (transposition).

I, D, R, T - штраф 1 (в преобразуемой строке).

Совпадение символа с самим собой не имеет дополнительной стоимости и оценивается как 0.

Эти операции и их стоимости позволяют вычислить расстояние Дамерау — Левенштейна между двумя строками, что является важным инструментом в областях, где требуется измерять сходство или различие между текстовыми последовательностями.

1.1 Нерекурсивный алгоритм поиска расстояния Левенштейна

Пусть $L1$ - длина строки S_1, $L2$ - длина строки S_2. $S_1[1..i]$ - подстрока S_1 с длиной i символов, начиная с первого, $S_2[1..j]$ - подстрока S_2 длиной j символов, начиная с первого. [2]

Расстояние Левенштейна может быть найдено с помощью формулы:

$$D(i,j) = \begin{cases} \max(i,j) & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j]), & \text{иначе} \\ \} & \end{cases}, \quad (1)$$

где $m(S_1[i], S_2[i])$ равна нулю, если $S_1[i] = S_2[i]$ и единице в противном случае.

Для оптимизации вычисления расстояния между строками используется матрица промежуточных значений. Её размерность определяется как $(len(S_1)+1)(len(S_2)+1)$. Каждая ячейка матрицы, обозначенная как $matrix[j,j]$, содержит значение расстояния между подстроками $S_1[1...i]S_2[1...j]$. Первая строка и первый столбец этой матрицы представляют собой тривиальные случаи и соответствуют наибольшим f i или j в соответствующей строке или столбце.

Матрица заполняется в соответствии со следующей формулой:

$$M[i][j] = min \begin{cases} M[i-1][j]+1, \\ M[i][j-1]+1, \\ M[i-1][j-1]+m(S_1[i],S_2[j]) \end{cases}. \quad (2)$$

Расстоянием Левенштейна будет значение в самой правой нижней ячейке матрицы с индексами $i = len(S_1), j = len(S_2)$.

1.2 Нерекурсивный алгоритм поиска расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна между двумя строками, обозначенными как S_1 и S_2, может быть вычислено с использованием формулы 3. Эта формула имеет следующий вид:

$$D(i,j) = \begin{cases} \min(i,j) = 0, & \text{если } \max(i,j), \\ \min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \end{cases} & \text{иначе} \\ \begin{cases} D(i-2,j-2)+1, & \text{если } i,j>1; \\ & S_1[i]=S_2[j-1]; \\ & S_1[i-1]=S_2[j] \\ \infty, & \text{иначе} \end{cases} \end{cases}, \quad (3)$$

Эта формула представляет собой рекуррентное соотношение, которое позволяет вычислить расстояние между двумя строками, учитывая вставки, удаления, замены и транспозиции символов. В результате выполнения этой формулы получается матрица, где значение в ячейке с индексами $i = len(S_1)$ и $j = len(S_2)$ представляет собой расстояние Дамерау — Левенштейна между исходными строками S_1 и S_2.

Формула 3 выводится на основе аналогичных рассмотрений, что и формула 1, но с учетом дополнительной редакторской операции - транспозиции символов. Эта формула позволяет эффективно находить расстояние Дамерау — Левенштейна и широко используется в задачах редактирования и сравнения строк.

1.3 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Рекурсивный алгоритм для вычисления расстояния Дамерау — Левенштейна реализует формулу 3. Функция D определяется следующим образом:

1) перевод из пустой строки в пустую строку не требует никаких операций и, следовательно, имеет стоимость равную нулю;
2) перевод из пустой строки в строку S_1 требует выполнения $|S_1|$ операций, каждая из которых добавляет один символ к строке S_1. Таким образом, общая стоимость такого перевода равна $|S_1|$;
3) перевод из строки S_1 в пустую строку аналогично требует выполнения $|S_1|$ операций удаления, каждая из которых удаляет один символ из строки S_1. Следовательно, общая стоимость такого перевода также равна $|S_1|$;
4) для перевода из строки S_1 в строку S_2 требуется выполнить последовательность операций (удаление, вставка, замена, транспозиция) в определенной последовательности.

Полагая S_1' и S_2' как строки S_1 и S_2 без их последних символов соответственно, цена преобразования из строки S_1 в строку S_2 может быть выражена следующими случаями:

а) сумма цены преобразования строки S_1' в S_2 и цены операции удаления, необходимой для преобразования S_1' в S_1;
б) сумма цены преобразования строки S_1 в S_2' и цены операции вставки, необходимой для преобразования S_2' в S_2;
в) сумма цены преобразования из S_1' в S_2' и цены операции замены, предполагая, что последние символы S_1 и S_2 разные;
г) цена преобразования из S_1' в S_2', предполагая, что последние символы S_1 и S_2 совпадают;
д) сумма цены преобразования S_1'' в S_2'', предполагая, что последние два символа S_1 можно преобразовать путем транспозиции в два последних символа S_2.

Таким образом, рекурсивный алгоритм находит оптимальное решение, учитывая все возможные варианты операций.

1.4 Рекурсивный алгоритм с кэшированием для нахождения расстояния Дамерау — Левенштейна

Рекурсивный алгоритм с кэшированием является улучшенной версией рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна. Он использует ту же формулу 3, но с добавлением механизма кэширования, чтобы избежать повторных вычислений и снизить вычислительную сложность.

Функция D в этой версии алгоритма хранит результаты вычислений в специальной матрице кэша. Каждый элемент этой матрицы соответствует значениям $D(i, j)$ для определенной пары индексов i и j. Если значение уже было вычислено, оно сохраняется в кэше, и при необходимости оно просто извлекается из кэша, вместо того чтобы пересчитываться заново.

Такой подход существенно ускоряет процесс вычисления расстояния Дамерау — Левенштейна, особенно при работе с большими строками. Эффективное использование кэширования позволяет избежать множественных повторных вычислений и сделать алгоритм более производительным.

Таким образом, рекурсивный алгоритм с кэшированием представляет собой оптимизированную версию рекурсивного алгоритма, способную эффективно находить расстояние Дамерау — Левенштейна между двумя строками.

2 Конструкторская часть

В этом разделе будут представлено описание используемых типов данных, а также схемы алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна.

2.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Схема нерекурсивного алгоритма нахождения расстояния Левенштейна представлена на рисунке 1.

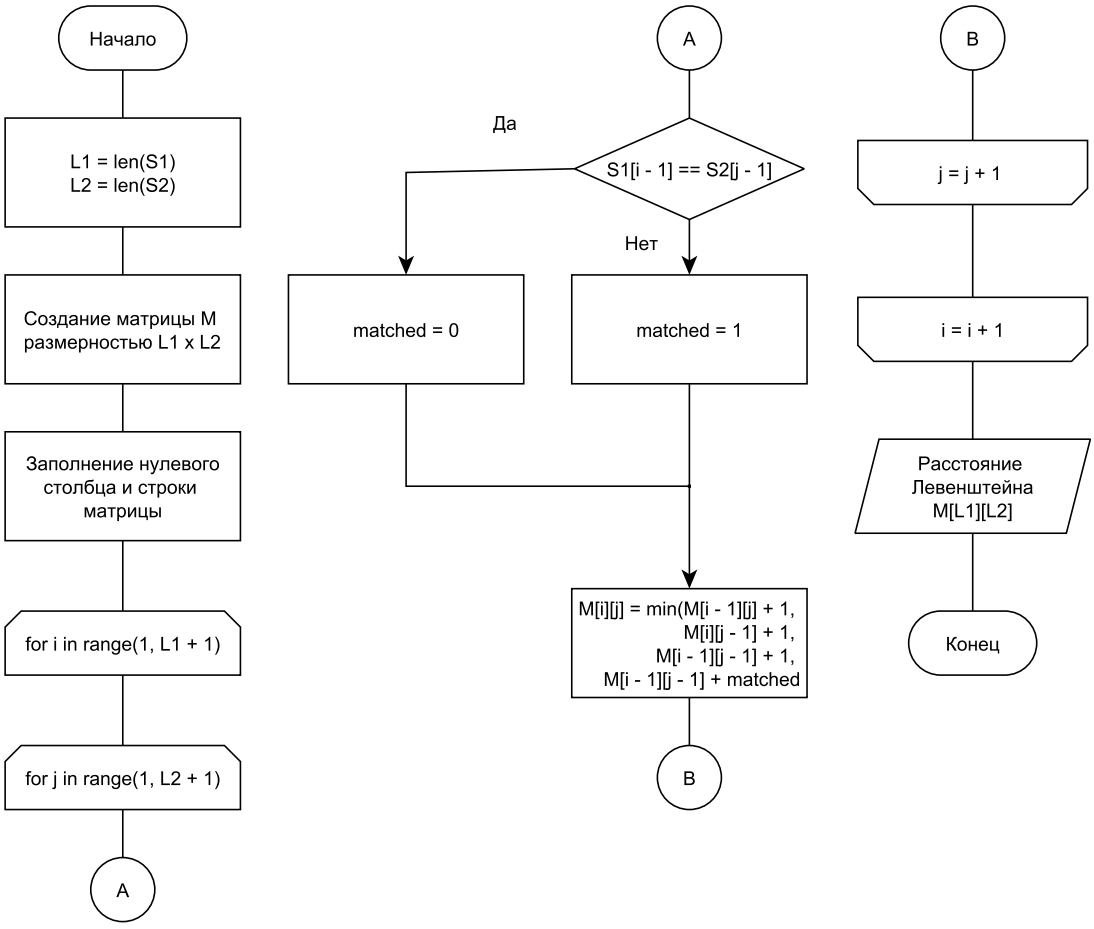

Рисунок 1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

2.2 Нерекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Схема нерекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна представлена на рисунке 2.

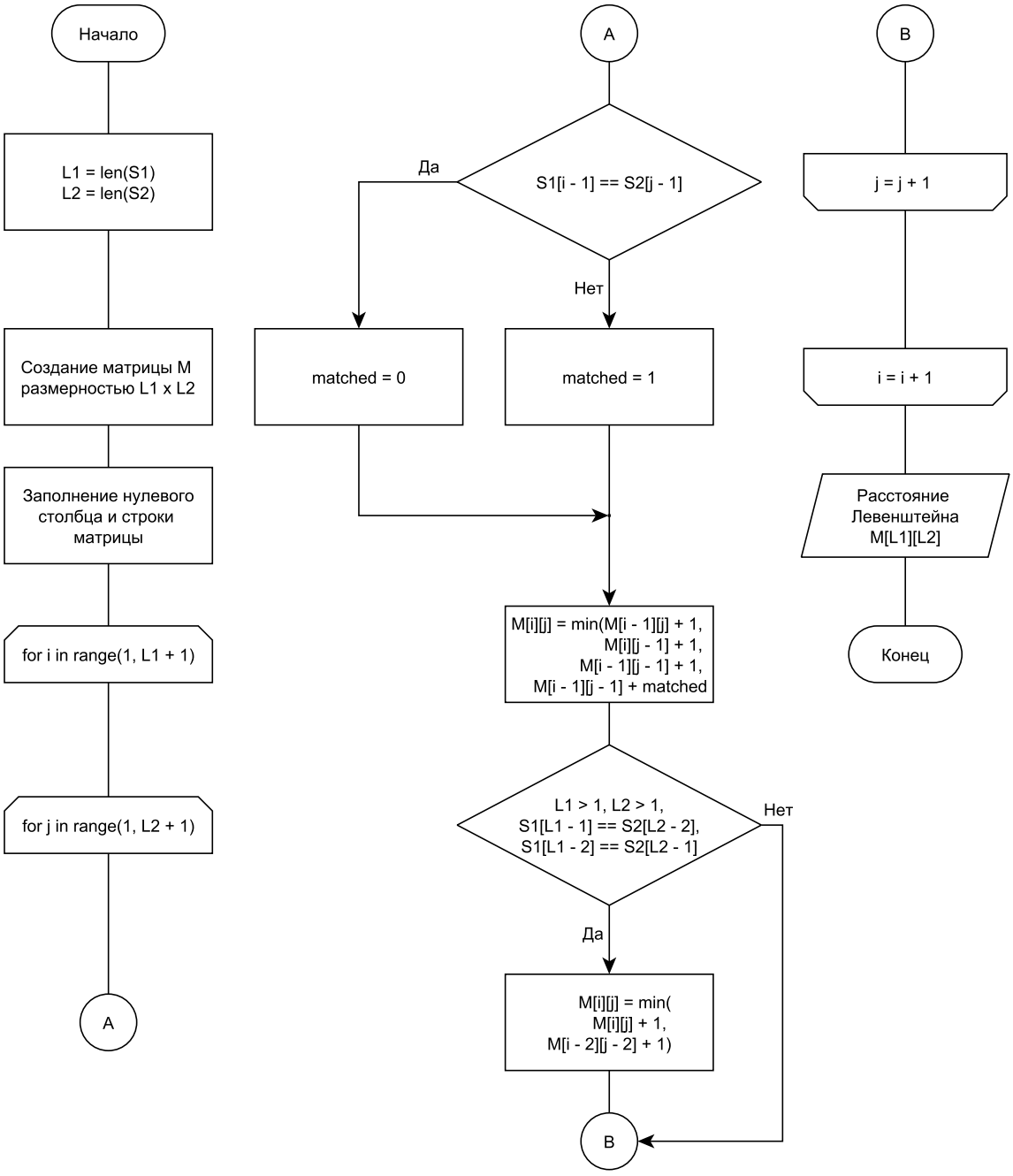

Рисунок 2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

2.3 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна представлена на рисунке 3.

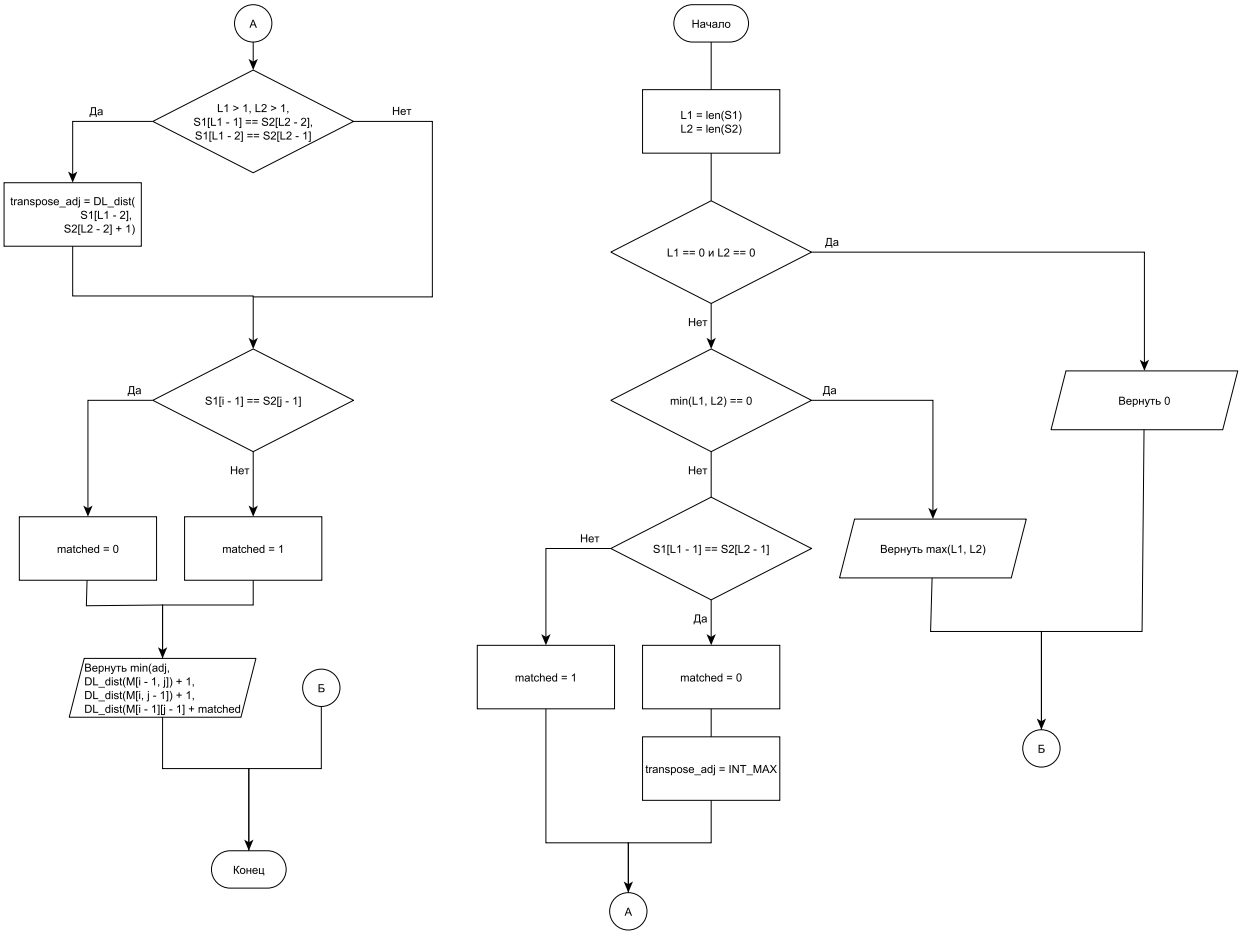

Рисунок 3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

]

2.4 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна с кэшем

Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с кэшем представлена на рисунке 4.

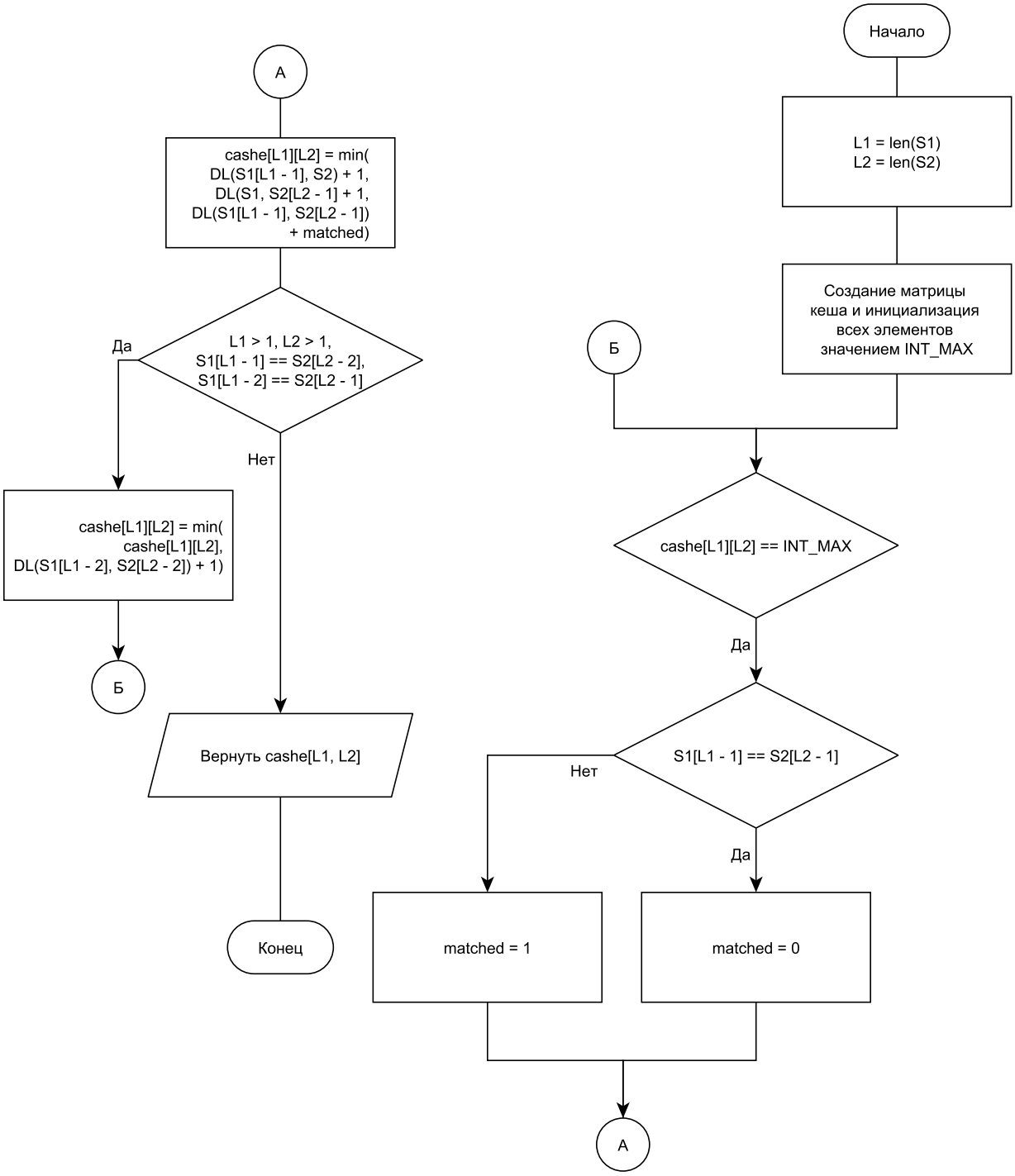

Рисунок 4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с кэшем

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются две строки на русском или английском языке в любом регистре;
- на выходе – искомое расстояние для трёх методов и их время выполнения.

3.2 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран C++.

Данный выбор обусловлен поддержкой языком парадигмы объектно – ориентированного программирования и наличием библиотек для точного замера времени.

Время выполнения реализации алгоритмов было замерено с помощью библиотеки *chrono*.

3.3 Сведения о модулях программы

Программа состоит из трех программных модулей:

1) main.cpp – главный модуль программы, содержащий функцию main и замер времени;
2) algorithm.cpp, algorithm.h – модуль с реализацией алгоритмов;
3) menu.cpp, menu.h – модуль с выбором режима работы программы;
4) tests.cpp, tests.h – модуль с тестированием программы.

3.4 Реализация алгоритмов

В листинге 1 приведена реализация итеративного алгоритма нахождения расстояния Левенштейна.

Листинг 1 – Итеративный алгоритм нахождения расстояния Левенштейна

```
int DistanceSolver::iterativeLevenshtein() {
    int L1 = S1.length();
    int L2 = S2.length();

    // Create and initialize a matrix of size (L1+1) x (L2+1)
    vector<vector<int>> dp(L1 + 1, vector<int>(L2 + 1, 0));

    // Initialize the first row and the first column of the matrix
    for (int i = 0; i <= L1; i++) {
        dp[i][0] = i;
    }
    for (int j = 0; j <= L2; j++) {
        dp[0][j] = j;
    }

    // Fill in the matrix
    for (int i = 1; i <= L1; i++) {
        for (int j = 1; j <= L2; j++) {
            int cost = (S1[i - 1] == S2[j - 1]) ? 0 : 1;
            dp[i][j] = min(min(dp[i - 1][j] + 1, dp[i][j - 1] + 1),
                dp[i - 1][j - 1] + cost);
        }
    }

    // The Levenshtein distance is found in the bottom-right corner
        of the matrix
    return dp[L1][L2];
}
```

В листинге 2 приведена реализация итеративного алгоритма нахождения расстояния Дамерау–Левенштейна.

Листинг 2 – Итеративный алгоритм нахождения расстояния Дамерау — Левенштейна

```
int DistanceSolver::iterativeDamerauLevenshtein() {
    int L1 = S1.length();
    int L2 = S2.length();

    // Create a 2D vector for dynamic programming
    vector<vector<int>> dp(L1 + 1, vector<int>(L2 + 1, 0));

    // Initialize the first row and column
    for (int i = 0; i <= L1; i++) {
        dp[i][0] = i;
    }
    for (int j = 0; j <= L2; j++) {
        dp[0][j] = j;
    }

    // Calculate the minimum edit distance using dynamic
       programming
    for (int i = 1; i <= L1; i++) {
        for (int j = 1; j <= L2; j++) {
            int cost = (S1[i - 1] == S2[j - 1]) ? 0 : 1;
            dp[i][j] = min(min(dp[i - 1][j] + 1, dp[i][j - 1] + 1),
                dp[i - 1][j - 1] + cost);

            // Check for character transposition
            if (i > 1 && j > 1 && S1[i - 1] == S2[j - 2] && S1[i -
               2] == S2[j - 1]) {
                dp[i][j] = min(dp[i][j], dp[i - 2][j - 2] + cost);
            }
        }
    }

    // Return the minimum edit distance
    return dp[L1][L2];
}
```

В листинге 3 приведена реализация рекурсивного алгоритма нахождения расстояния Дамерау–Левенштейна.

Листинг 3 – Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

```
int DistanceSolver::recursiveDamerauLevenshtein(int L1, int L2) {
    // Base case: If both strings are empty, return 0 (edit distance)
    if (L1 == 0 && L2 == 0) return 0;

    // If one of the strings is empty, return the length of the other string
    if (L1 == 0) return L2;
    if (L2 == 0) return L1;

    // Calculate the cost for the current character pair
    int cost = (S1[L1 - 1] == S2[L2 - 1]) ? 0 : 1;

    // Calculate the edit distances for insertion, deletion, and substitution
    int insertion = recursiveDamerauLevenshtein(L1, L2 - 1) + 1;
    int deletion = recursiveDamerauLevenshtein(L1 - 1, L2) + 1;
    int substitution = recursiveDamerauLevenshtein(L1 - 1, L2 - 1) + cost;

    // Find the minimum edit distance among the three operations
    int result = min(min(insertion, deletion), substitution);

    // Check for character transposition
    if (L1 > 1 && L2 > 1 && S1[L1 - 1] == S2[L2 - 2] && S1[L1 - 2] == S2[L2 - 1]) {
        int transposition = recursiveDamerauLevenshtein(L1 - 2, L2 - 2) + cost;
        result = min(result, transposition);
    }

    return result;
}
```

В листинге 4 приведена реализация рекурсивного алгоритма нахождения расстояния Дамерау–Левенштейна с кэшем.

Листинг 4 – Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна с кэшем

```
int DistanceSolver::recursiveCacheDamerauLevenshtein(int L1, int L2) {
    if (cache[L1][L2] != INT_MAX) {
        return cache[L1][L2];
    }
    if (L1 == 0) {
        cache[L1][L2] = L2;
        return L2;
    }
    if (L2 == 0) {
        cache[L1][L2] = L1;
        return L1;
    }

    int cost = (S1[L1 - 1] == S2[L2 - 1]) ? 0 : 1;
    int insertion = recursiveCacheDamerauLevenshtein(L1, L2 - 1) + 1;
    int deletion = recursiveCacheDamerauLevenshtein(L1 - 1, L2) + 1;
    int substitution = recursiveCacheDamerauLevenshtein(L1 - 1, L2 - 1) + cost;
    int result = min(min(insertion, deletion), substitution);

    if (L1 > 1 && L2 > 1 && S1[L1 - 1] == S2[L2 - 2] && S1[L1 - 2] == S2[L2 - 1]) {
        int transposition = recursiveCacheDamerauLevenshtein(L1 - 2, L2 - 2) + cost;
        result = min(result, transposition);
    }

    cache[L1][L2] = result;
    return result;
}
```

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялся замерный эксперимент:

— операционная система Windows 11;

— память 16 ГБ;

— процессор 3,6 ГГц 6-ядерный процессор AMD Ryzen 5000 series 5.

Замеры проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только интегрированной средой разработки и непосредственно выполняемой программой.

4.2 Пример работы программы

На рисунке 5 представлен пример работы программы. Вводятся две строки. Для этих строк высчитывается расстояние Левенштейна и Дамерау-Левенштейна. Выводится результат вычислений и время работы каждого алгоритма в микросекундах.

```
Введите первую строку S1: abacaba
Введите вторую строку S2: ababaga
Расстояние Левенштейна: 2
Расстояние Дамерау-Левенштейна: 2
Левенштейн: 4.746000e+03 тактов
Дамерау-Левенштейн: 7.770000e+03 тактов
Рекурсивный Дамерау-Левенштейн: 8.531670e+05 тактов
Рекурсивный Дамерау-Левенштейн с кэшем: 3.234000e+03 тактов
```

Рисунок 5 – Пример работы программы

4.3 Время выполнения реализованных алгоритмов

Введём следующие обозначения:

Л - Левенштейна

ДЛ - Дамерау — Левенштейн

РДЛ - Рекурсивный Дамерау — Левенштейн

РДЛК - Рекурсивный Дамерау — Левенштейн с кэшем

Таблица 1 – Результаты замеров времени реализованных алгоритмов в тактах процессора

Длина строк	Л	ДЛ	РДЛ	РДЛК
1	1.720e3	2.014e3	8.726e1	1.148e2
2	1.483e3	1.759e3	2.359e1	2.646e1
3	2.797e3	2.803e3	1.462e2	6.205e1
4	3.084e3	2.897e3	7.193e2	1.030e2
5	4.516e3	4.042e3	3.699e3	2.098e3
6	3.984e3	3.984e3	1.729e4	2.011e4
7	3.785e3	3.513e3	8.308e4	2.490e4
8	6.053e3	5.256e3	5.534e5	6.769e3
9	8.088e3	5.973e3	2.640e4	8.404e3
10	8.570e3	6.942e3	1.433e5	1.107e5

Замеры времени работы реализованных алгоритмов для определенной длины строк проводились 100 раз, при этом каждый раз строки генерировались случайно. Для измерения тактового времени была использована инструкция rdtsc [3]. В качестве результата, представленного в таблице 1, взято среднее время на каждой длине слова.

4.4 Занимаемая память реализованных алгоритмов

В данном разделе представлены результаты замеров затрат памяти для различных алгоритмов на обработку строк разной длины. Таблица 2 демонстрирует объем памяти в байтах, потребляемый разными реализациями алгоритмов в зависимости от длины обрабатываемых строк.

Таблица 2 – Затраты памяти в байтах для различных алгоритмов в зависимости от длины строк

N	Л	ДЛ	РДЛ	РДЛК
1	28	28	32	64
2	48	48	32	140
3	76	76	32	224
4	112	112	32	352
5	156	156	32	544
6	208	208	32	800
7	268	268	32	1088
8	336	336	32	1376
9	412	412	32	1696
10	496	496	32	2032

4.5 Вывод

В результате анализа замеров времени выполнения и затрат памяти на различных алгоритмах были сделаны следующие выводы:

- для обработки строк длины меньшей, чем пять символов, рекомендуется использовать рекурсивные алгоритмы;
- для строк, содержащих больше пяти символов, рекомендуется использовать итеративные алгоритмы, т.к. они предоставляют лучшую производительность;
- при необходимости учета транспозиций символов алгоритм Дамерау — Левенштейна с кэшированием (РДЛК) является предпочтительным выбором.

ЗАКЛЮЧЕНИЕ

Цель работы достигнута: изучены алгоритмы нахождения расстояния Левенштейна и Дамерау — Левенштейна.

В ходе выполнения лабораторной работы были решены все задачи:

- изучены алгоритмы нахождения расстояния Левенштейна и Дамерау — Левенштейна;
- применены методы динамического программирования для реализации алгоритмов;
- на основе полученных в ходе экспериментов данных были сделаны выводы по поводу эффективности всех реализованных алгоритмов;
- был подготовлен отчет по лабораторной работе.

Эксперименты показали, что наиболее затратный по времени рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна без кеша, а наименее затратны итеративные алгоритмы. Менее затратными по памяти являются реализации итеративных алгоритмов. Самым затратным по памяти является реализация рекурсивного алгоритма поиска расстояния Дамерау — Левенштейна с кешированием.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Модифицированный алгоритм исправления ошибок в информационно-поисковых запросах / М. В. Алдошин [и др.] // Известия Тульского государственного университета. Технические науки. — 2020. — С. 5—8. — URL: https://cyberleninka.ru/article/n/modifitsirovannyy-algoritm-ispravleniya-oshibok-v-informatsionno-poiskovyh-zaprosah.
2. *Левенштейн В. И.* Двоичные коды с исправлением выпадений, вставок и замещений символов. Т. 163. — Москва : Доклады АН СССР, 1965. — С. 845—848.
3. *Microsoft.* RDTSC (Read Time-Stamp Counter). — URL: https://learn.microsoft.com/ru-ru/cpp/intrinsics/rdtsc?view=msvc-170.